Mobile Agents – The Right Vehicle for Distributed Sequential Computing

Lei Pan, Lubomir F. Bic, Michael B. Dillencourt, and Ming Kin Lai

Information and Computer Science, University of California
Irvine, CA 92697-3425
{pan,bic,dillenco,mingl}@ics.uci.edu

Abstract. Distributed sequential computing (DSC) is computing with distributed data using a single locus of computation. In this paper we argue that computation mobility—the ability for the locus of computation to migrate across distributed memories and continue the computation as it meets the required data—facilitated by mobile agents with strong mobility is essential for scalable distributed sequential programs that preserve the integrity of the original algorithm.

Keywords: distributed sequential computing (DSC), computation mobility, mobile agents, distributed code building block (DBlock), scalability, algorithmic integrity, paging, Crout factorization

1 Introduction

Sequential computing will never disappear, even in distributed environments. There are at least three reasons for this. First, only very few algorithms (referred to as "embarrassingly parallel") can be perfectly parallelized. In fact, Amdahl's law tells us that the sequential portions in a parallel program often become bottlenecks for speedup, so we need to pay special attention to them. Second, when solving problems on distributed-memory systems, programmers may choose to decompose a problem into coarse-grained sequential sub-tasks that run in parallel, even if fine-grained data parallelism is an option [1]. Third, although parallel compilers can sometimes do a good job of translating sequential code to parallel code, not all auto-translations are viable and in many cases human intervention is unavoidable [2]. Re-implementing an algorithm in parallel can require a major programming effort. In some cases, significant performance improvement can be achieved without this additional programming effort by running the sequential algorithm in a distributed environment to solve large problems and avoid the performance penalty from disk thrashing [3, 4, 5]. We define *distributed sequential computing* (DSC) as computing with distributed data using a single locus of computation.

The key to scalability in distributed high performance computing is reducing the amount of data being moved. The classical distributed shared memory (DSM) systems cannot address this issue efficiently and nicely, therefore they cannot fulfill the initial expectations after almost 15 years of research [6].

Not willing to give up the convenience of "reading and writing remote memory with simple assignment statements" [7], researchers tried to exploit data locality in later improved DSM-based systems or compilers, such as HPF [8, 9] and UPC [7, 10], by following the general principle of "owner-computes" [11]. The principle states that computations should happen on the processor that owns the value that is being assigned to. The use of this principle in HPF or UPC is somewhat limited: a programmer can only exercise "owner-computes" by first specifying data distribution pattern and then using new language constructs such as **forall**. Strict adherence to the principle of "owner-computes" as articulated above does not always minimize data movement. For example, in an assignment, if the right-hand-side (RHS) objects are large in size and distributed, instead of all the RHS objects being sent to the owner of the left-hand-side (LHS) object for computation, the result could instead be computed on the home processor of the RHS objects and then sent to the owner of the LHS object for assignment. This would reduce the amount of communication required. In this paper, we will reuse the term "owner-computes" but extend the meaning to be "owner process/thread of large sized data computes" and the scope to be code building blocks that can have one or more assignments.

The goal of "owner-computes" in our extended sense is to minimize communication cost. The message passing (MP) approach, as exemplified by MPI, follows the rule strictly, which is why it is efficient and scalable, and hence popular in the high-performance commercial software market [12]. However, MP has moved a step too far, which makes it hard to use. MP is "too local" in two ways. First, MP programmers usually have only a "local view" of the distributed data, rather than a global view from which the "original algorithm" (the sequential or PRAM [13] algorithm) is developed [1]. This is because the data reference (e.g., array indexing) is local to a distributed data piece that is owned by a process. Second, moving the locus of computation from one "local process" to another in MP is cumbersome since it usually requires artificial constructs and synchronization. The inconsistency in data view and the additional programming required to support the transfer of computation locus can cause the MP implementation to look dramatically different from the original algorithm.

Moving computation locus across memory boundaries is unavoidable if we follow the rule of "owner-computes" in distributed memory. This observation is an immediate consequence of the following two basic facts: (1) in order for a computation to be performed on a data item, the data item and the locus of computation need to be together; and (2) the "owners" of distributed data pieces change across memory boundaries. We define *computation mobility* as the ability for the locus of computation to migrate across distributed memories and continue the computation as it meets the required data. This migration is controlled by a programmer either explicitly or implicitly through data distribution. Mobile agents that provide strong mobility [14] are a means to facilitate computation mobility. If the computation mobility in a distributed sequential program is implemented using a mobile agent system, we call this type of computing mobile-agent-based DSC. One subtle but important point is that mobile-agent-based computation mobility does not necessarily mean code has to move. In fact, a careful implementation of the underlying agent infrastructure allows code to either be loaded from a shared disk or, in a non-shared file system, to be sent across the network at most once, irrespective of how many times the locus of computation moves across the network [15]. This is crucial to performance. Strong mobility in our mobile agent system means that execution state but not always code is allowed to migrate.

One immediate application of DSC is to utilize the power of a network of workstations to improve the performance of data-intensive sequential programs without re-programming them. This is based on the observation that under certain circumstances partitioning the data onto different machines and reducing the disk paging overhead by using the collective memory of a network of workstations can result in considerable performance increase, without converting the underlying algorithm to a parallel implementation. One way of utilizing the distributed memories is the "remote memory paging" or "network RamDisk" approach [4, 5], a special case of DSM, in which a process runs on a single machine and accesses data remotely. A major disadvantage of this approach is its nonscalability because the rule of "owner-computes" is clearly violated here. Another approach is to use the mobile-agent-based DSC approach [3]. The data is distributed over the workstations in the network just as in the "remote memory paging" approach. The difference is that rather than having the program run on a single machine and remotely access the data, the computation, using a mobile agent as a vehicle, moves to the data. As computation locus, carried by a mobile agent, moves to each piece of the distributed data, it dynamically becomes the owner of large sized data and therefore the rule of "owner-computes" is always followed.

In this paper, we argue that computation mobility facilitated by mobile agents with strong mobility is not only *a good way* to implement DSC, but also *the right way*. Of course, anything that can be done with mobile agents can be done with MP: after all, at low level mobile agents are ultimately streams of bytes and hence messages. But mobile-agent-based DSC provides a new layer of abstraction that helps to improve programmability in two ways. First, a DSC program implemented as a mobile agent preserves a global view of the problem data through shared variable programming [16]. The distributed data is bridged by mobile agents at the application level. Second, mobile-agent-based DSC eliminates the need of manually adding artificial auxiliary threads, and handling state transfer and synchronization among the real and the auxiliary threads. This observation, that mobile agents simplify the programming task by eliminating the necessity of explicitly maintaining the state of the processes or threads, has been previously articulated elsewhere [17].

The rest of the paper is organized as follows. Section 2 compares, using a simple and abstract example, the different ways mobile agents, MP, and DSM would bring computation locus and data together sequentially in distributed memory. Section 3 describes a real application, Crout factorization, and its different implementations. The last section contains some final remarks.

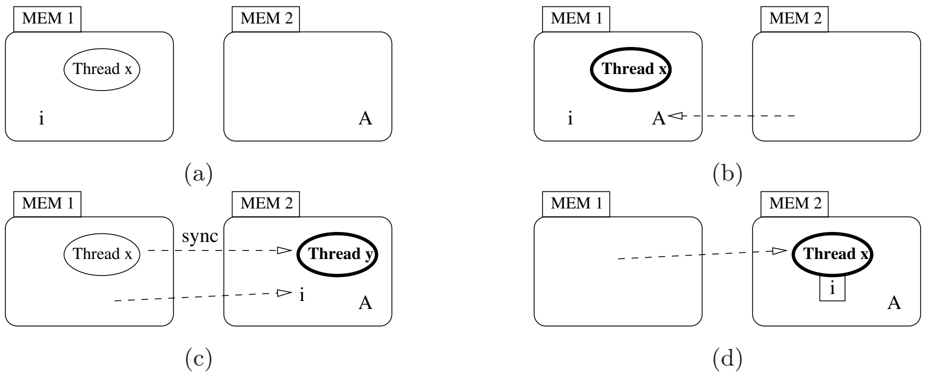

Fig. 1. Data and computation rendezvous. (a) The problem. (b) The classical DSM approach. (c) The MP or improved DSM approach. (d) The mobile agent approach

2 Bringing Together Data and Computation

A sequential program consists of code building blocks, which may be nested or may follow one another sequentially. We use the notation $\mathcal{B}_{\mathrm{T}}(\mathcal{D})$ to denote a code building block of type T that performs its computation on a collection of data pieces denoted by $\mathcal{D}$. The type T of a code building block can be any of the basic programming constructs such as a loop, an **if** statement, a multi-way conditional statement (e.g., a **switch**/**case** block), or a sequence of statements. The collection of data pieces $\mathcal{D}$ represents the data (input, output, intermediate data) used in the execution of the code building block. When the data pieces in the collection $\mathcal{D}$ are distributed over disjoint memories, $\mathcal{B}_{\mathrm{T}}(\mathcal{D})$ is called a *distributed code building block*, or a *DBlock*. DBlocks are the natural constructs to consider when performing sequential computation using distributed data; indeed, converting code building blocks into DBlocks is the fundamental problem that must be solved when turning a sequential program into a distributed sequential program. As the problem scales up, more code building blocks in a program become DBlocks, and the data in DBlocks are distributed across more memories. Hence the ability to handle DBlocks in an efficient and consistent way is the key to scalable distributed sequential computing.

Consider a DBlock $\mathcal{B}_{\mathrm{T}}(i, A)$, where i is a relatively small data piece and A is large. A and i could be from different variables (e.g., if i is the data stored in a loop index variable and A is a matrix or a portion thereof), or they could be from the same variable (e.g., if i is a column of a matrix and A is a portion of the same matrix that consists of a large number of columns). Suppose that the data pieces i and A are distributed as shown in Fig. 1(a). $\mathcal{B}_{\mathrm{T}}(i, A)$, executed by Thread x, needs to bring together these two data pieces to continue the computation. There are several ways of doing this. These are shown in Fig. 1(b)-(d), where in each case the highlighted thread is the one that continues the computation, and the dotted arrows indicate a transfer of locus of computation, data, or synchronization. The classical DSM approach is shown in Fig. 1(b). Thread x, which is stationary, pulls data A to meet with data i and itself. This causes more data than necessary to be moved, and violates the rule of "owner-computes."

The approach used by MP and by improved DSM systems such as HPF or UPC is illustrated in Fig. 1(c). Since only stationary threads are available, the programmer creates an artificial auxiliary process or thread, Thread y in the figure, to handle the computation after the transfer of the computation locus. The transfer of the computation locus from Thread x to Thread y requires data passing and synchronization. In some cases, the data i can be redundantly computed on all processors, eliminating explicit data passing but requiring each thread to create and maintain additional state data, and an explicit synchronization. (For example, a loop can be run on all processors to redundantly compute the value i, and an artificial mask, in the form of an **if** statement, can be used to identify the owner process of data A. This is the SPMD programming style; the **if** statements are the way SPMD creates auxiliary threads.) In other cases, synchronization can be performed implicitly via explicit data passing. It is worth noting that the **forall** construct in HPF or UPC is designed for data parallel loops (i.e., loops in which iterations can be done concurrently) rather than for constructing a sequential loop that spans distributed memories.

The mobile agent approach to implementing a DBlock is shown in Fig. 1(d). The thread performing the computation "bundles" the data i into a local agent variable, which it then carries to MEM 2 to perform the computation. This is not only efficient (because the agent thread becomes the owner of the large data A before it computes, without moving this data) but also natural to a programmer, because the rendezvous of computation locus and data is seamless.

At the level of abstraction appropriate for algorithm design, a DBlock is no different from a non-distributed block: the data is manipulated by the single computation locus. The requirement that computation and data be brought together is imposed by the distributed environment and the size of the problem being solved, since data is distributed across disjoint memories as the problem scales up. Thus the integrity of the algorithm is best preserved by bringing computation and data together via intra-thread data carrying rather than inter-thread data passing. Mobile agents with strong mobility provide a layer of abstraction that helps this to happen in a seamless and consistent fashion. At a low level, mobile agent code is generally compiled down to stationary threads passing data between them. By programming at the mobile agent level rather than the MP level and letting a compiler do the translation, the programmer avoids a tedious, time-consuming, and error-prone translation task. In the next section, we will provide an example illustrating how programming at this level of abstraction can help with a real-world application.

```
(1)   For $j = 1$ to $N$
(2)      For $i = 2$ to $j - 1$
(3)         $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$
(4)      End For
(5)      For $i = 1$ to $j - 1$
(6)         $T \leftarrow K_{ij}$
(7)         $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(8)         $K_{jj} \leftarrow K_{jj} - TK_{ij}$
(9)      End For
(10)  End For
```

(a)

```
(1)      For $j = 1$ to $N$
(1.1)       load column $j$
(2)         For $i = 2$ to $j - 1$
(2.1)          hop to column $i$
(2.2)          load $K_{ii}$
(3)            $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$
(4)         End For

(4.1)       hop to column $j$
(4.2)       unload column $j$

(5)         For $i = 1$ to $j - 1$
(6)            $T \leftarrow K_{ij}$
(7)            $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(8)            $K_{jj} \leftarrow K_{jj} - TK_{ij}$
(9)         End For
(10)     End For
```

(b)

Fig. 2. Pseudocode for Crout factorization. (a) The sequential implementation. (b) The mobile-agent-based DSC implementation

3 Distributed Sequential Crout Factorization

In this section, we describe one example, Crout Factorization, for which performance data is provided in our other paper [3]; here the focus is on why, once we have decided to use DSC, the mobile-agent-based DSC approach is better than MP. In essence, Crout Factorization is a method of factoring a symmetric positive-definite $N \times N$ matrix K into the product of three matrices $K = U^T DU$, where U is an upper triangular matrix with unit diagonal entries and D is a diagonal matrix. Typically, K is a sparse banded matrix, meaning that entries that are more than a fixed distance b, called the *half-bandwidth*, from the diagonal are 0. Figure 2(a) shows the pseudocode for Crout factorization. In line (3), the summation over l corresponds to a dot product of two sub-vectors of columns i and j. These are the two shaded vectors in Fig. 3(a). The computation of column j depends on previously computed columns. The "working set" of matrix entries required to compute column j is shown shaded in Fig. 3(b).

When the size of the working set exceeds the size of the main memory on a single workstation, extensive paging overhead occurs. This thrashing can be eliminated by using the mobile-agent-based DSC implementation of the algorithm [3]. The idea is to split the matrix into pieces, where each piece is a contiguous set of columns. The size of the piece is chosen so that each piece can fit into the main memory of one workstation. The algorithm runs on P workstations, where P is the number of pieces that comprise K. Figure 3(c) shows an

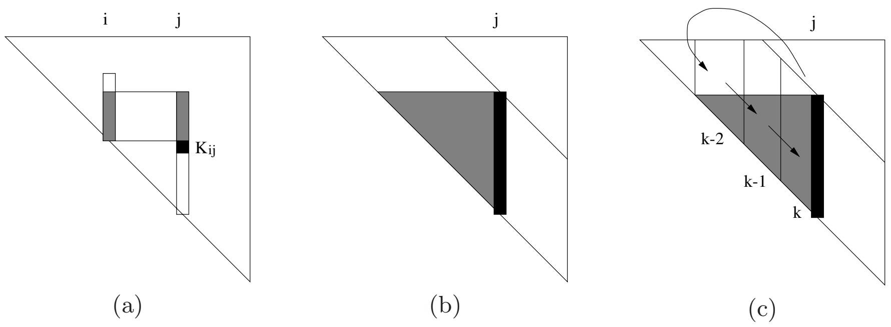

Fig. 3. Crout factorization: (a) Computing of K_{ij} requires the dot product of the two shaded vectors. (b) Working set for column j. (c) Working set decomposition

example for which the working set is subdivided into three pieces. The arrows indicate how an agent, carrying column j which it is computing, would move through the pieces of the working set.

When the size of the working set, which is problem dependent, is too large for a single workstation, pulling the entire working set to a single stationary process, as done in the "remote memory paging" or DSM approach, would not only require much more data to be transferred, but also cause "remote memory thrashing" instead of "local disk thrashing," if "least recently used" is the underlying protocol for handling paging. This is because the data access pattern, shown with arrows in Fig. 3(c), is such that columns are paged out of the (local) main memory right before they are going to be used.

The mobile-agent-based DSC implementation of Crout factorization is shown in Fig. 2(b). The only difference between this code and the sequential code is that two hop statements and three load/unload statements are inserted. These statements are navigational annotations telling the computation which node to hop to (given a column index) and what data to load or unload. They do not modify the structure of the existing sequential algorithm; in other words, they preserve algorithmic integrity. Although there are a large number of hops, most of them will be local and hence will be no-ops with negligible cost. The load statements involve copying a single column (at line (1.1)) or a single matrix entry (at line (2.2)) into agent variables. Once the new values of column j have been computed, by the agent visiting the nodes that contain the pieces of the working set, they are unloaded, or copied back into the appropriate location on the node storing column j by the unload statement at line (4.2).

There are two DBlocks that we focus on in our implementation shown in Fig. 2(b). The first one is $\mathcal{B}_{\text{for}}(\{j\}, \{K_{lj}\})$ which includes lines (1)–(10). In this DBlock, the data $\{K_{lj}\}$ is column j of matrix K which is much larger in size than data $\{j\}$ which holds the loop index variable. The variable j is an agent variable, thus the data $\{j\}$ can follow the computation locus and meet column j wherever the column resides. The second DBlock is $\mathcal{B}_{\text{for}}(\{i\}, \{j\}, \{K_{lj}\}, \{K_{ii}\}, \{K_{li}\})$ which consists of lines (2)–(4) (the loop that computes *dot product* of columns j

```
(1)  For $k = 1$ to $P$
(2)    If ($\mu == k$) Then
(3)      For $j = I_k$ to $I_{k+1} - 1$
(4)        Send ($k - 2$, col $j$)
(5)        Recv ($k - 1$, col $j$, $\{K_{dd}\}$)
(6)        For $i = I_k$ to $j - 1$
(7)          $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(8)        End For
(9)        For $i = 1$ to $j - 1$
(10)         $T \leftarrow K_{ij}$
(11)         $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(12)         $K_{jj} \leftarrow K_{jj} - T K_{ij}$
(13)       End For
(14)     End For
(15)   Else If ($\mu == k - 2$) Then
(16)     For $m = I_k$ to $I_{k+1} - 1$
(17)       Recv ($k$, col $j$)
(18)       For $i = I_{k-2}$ to $I_{k-1} - 1$
(19)         $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(20)       End For
(21)       Send ($k - 1$, col $j$, $\{K_{dd}\}$)
(22)     End For
(23)   Else If ($\mu == k - 1$) Then
(24)     For $m = I_k$ to $I_{k+1} - 1$
(25)       Recv ($k - 2$, col $j$, $\{K_{dd}\}$)
(26)       For $i = I_{k-1}$ to $I_k - 1$
(27)         $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(28)       End For
(29)       Send ($k$, col $j$, $\{K_{dd}\}$)
(30)     End For
(31)   End If
(32) End For
```

Fig. 4. Pseudocode for Crout factorization using MP or improved DSM

and i). In this DBlock, the data $\{i\}$ is the value of the loop index variable i, an agent variable similar to variable j. The data $\{K_{lj}\}$ is a copy of column j in an agent variable represented by K_{ij} and K_{lj} (at line (3)). The data $\{K_{ii}\}$ is the diagonal entries of K loaded in an agent variable K_{ii} (at line (2.2)). The data $\{K_{li}\}$ is the working set of column j. It is the largest sized data in this DBlock, and is distributed in node variable K_{li}. In the second DBlock, the data pieces $\{i\}$, $\{j\}$, $\{K_{lj}\}$, and $\{K_{ii}\}$ are carried to meet with $\{K_{li}\}$. Lines (5)–(9) (the loop that does *scaling* of column j with diagonal entries $\{K_{ii}\}$) make a code building block that would run on one workstation, for each value of j, with all data local, and hence they do not make a DBlock. In this code building block, K_{ij} and K_{jj} are node variables, and K_{ii} is the same agent variable as the one in line (2.2).

It is of course possible to implement Crout factorization using MP-based DSC. Figure 4 shows the pseudocode. Notice that this pseudocode is only at a high level, and some details, e.g., the boundary cases of k, are left out. In the pseudocode, μ is the process ID, which is defined as the index of a matrix piece that a processor owns. I_k is the index of the first column that piece k owns. And $\{K_{dd}\}$ is a vector of diagonal entries. If we compare the MP implementation shown in Fig. 4, with the original algorithm shown in Fig. 2(a), the differences are considerable. Producing the MP version requires carefully analyzing and explicitly handling the "roles," or states of various processes using artificial **if** masks. These masks artificially cut a DBlock into sub- code building blocks that belong to different processes. These sub-blocks are each code building blocks, but no longer DBlocks, because they only work on data that is local (the received messages are buffered locally). In Fig. 4, code lines (6)–(8), (18)–(20), and (26)–(28) used to belong to the same code building block in the original algorithm (lines (2)–(4) in Fig. 2(a)), or the agent code (lines (2)–(4) in Fig. 2(b)), but they are broken up by MP into different sub-blocks, as the original code building block grows with the problem size to become DBlocks. In contrast, mobile-agent-based DSC implementation handles this block transition seamlessly and consistently through intra-agent data carrying. A local view of distributed data pieces is used in MP, which is reflected by the fact that each process only runs loops over the columns it owns, as it updates only its own data. Both the local data view and broken DBlocks in MP implementation contribute to the change of the high-level structure of the original algorithm, which significantly complicates the task of new code development as well as old code maintenance.

4 Final Remarks

We have demonstrated in this paper that DSC is a natural fit with mobile agents and a very poor fit with MP, which is a lower level approach to distributed computing. As mentioned in the introduction, one reason why DSC is important is that a parallel problem using distributed data can be decomposed into a collection of cooperating subtasks, each implemented using DSC. An example of this is presented in a companion paper [18].

One advantage of DSM is that it supports incremental parallelization of sequential programs [1]. This is because sequential programs can be ported to a DSM system without much efforts. Once this initial porting is complete, the programmer can incrementally re-implement portions in parallel. Our mobile-agent-based distributed computing offers the same advantage for essentially the same reason: because of algorithmic integrity, transforming a sequential program to a DSC program is straightforward.

The introduction of DBlocks in Section 2 gives some insight into the nature of the programming tasks using DSM, MP, or our mobile-agent-based approach. With classical DSM, DBlocks are handled completely transparently as if they were not distributed blocks at all. This is extremely convenient, but it comes at a steep price: scalability is lost because the “owner-computes” rule is violated. In MP or improved DSM, DBlocks are broken into code building blocks belonging to different processes. The “owner-computes” rule is followed, but the original code structure is changed significantly. In our approach, blocks that are not distributed are coded exactly as they are in the original code, while DBlocks are annotated with navigational commands but otherwise preserve their code structure. As the problem size increases, more code building blocks are turned into DBlocks, and more navigational commands are therefore inserted. This provides a natural migration path along which programs can evolve to solve increasingly large problems.

Acknowledgment

The authors wish to thank Prof. Keith Marzullo and the anonymous reviewers for their incisive comments on an earlier draft of this paper.

References

[1] Leopold, C.: Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches. John Wiley & Sons, Inc. (2001) 575, 576, 583

[2] Berthou, J. Y., Colombet, L.: Which approach to parallelizing scientific codes—that is the question. Parallel Computing **23** (1997) 165–179 575

[3] Pan, L., Bic, L. F., Dillencourt, M. B.: Distributed sequential computing using mobile code: moving computation to data. In: ICPP2001: 30th International Conference on Parallel Processing, Valencia, Spain, IEEE (2001) 77–86 575, 577, 580

[4] Dramamitos, G., Marktos, E. P.: Adaptive and reliable paging to remote main memory. Journal of Parallel and Distributed Computing **58** (1999) 357–388 575, 577

[5] Flouris, M. D., Markatos, E. P.: The network ramdisk : Using remote memory on heterogeneous NOWs. Cluster Computing **2** (1999) 281–293 575, 577

[6] Tanenbaum, A. S., Van Steen, M.: Distributed Systems Principles and Paradigms. Prentice Hall, Upper Saddle River, NJ 07458 (2002) 575

[7] Carlson, W. W., Draper, J. M., Culler, D. E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and language specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences (1999) 576

[8] Merlin, J., Hey, A.: An introduction to High Performance Fortran. Scientific Programming **4** (1995) 87–113 576

[9] Schreiber, R. S.: An introduction to HPF. Lecture Notes in Computer Science **1132** (1996) 27–44 576

[10] El-Ghazawi, T., Chauvin, S.: UPC benchmarking issues. In: 2001 International Conference on Parallel Processing (ICPP '01), Valencia, Spain, IEEE (2001) 365–372 576

[11] Rogers, A., Pingali, K.: Compiling for distributed memory architectures. IEEE Transactions on Parallel and Distributed Systems **5** (1994) 281–298 576

[12] Gropp, W. D.: Learning from the success of MPI. Lecture Notes in Computer Science **2228** (2001) 81–92 576

[13] Keller, J., Kebler, C. W., Traff, J. L.: Practical PRAM Programming. Wiley, New York (2000) 576

[14] Fuggetta, A., Picco, G. P., Vigna, G.: Understanding Code Mobility. IEEE Transactions on Software Engineering **24** (1998) 342–361 576

[15] Gendelman, E., Bic, L. F., Dillencourt, M. B.: Fast file access for fast agents. In: MA 2001: 5th International Conference on Mobile Agents, Atlanta, Georgia (2001) 88–102 577

[16] Pan, L., Bic, L. F., Dillencourt, M. B.: Shared variable programming beyond shared memory: Bridging distributed memory with mobile agents. In: IDPT2002: The Sixth International Conference on Integrated Design and Process Technology, Pasadena, CA (2002) 577

[17] Chess, D., Harrison, C., Kershenbaum, A.: Mobile agents: Are they a good idea? In: 2nd Int. Workshop on Mobile Object Systems, Springer LNCS 1222 (1997) 25–47 577

[18] Pan, L., Bic, L. F., Dillencourt, M. B., Huseynov, J. J., Lai, M. K.: Distributed parallel computing using navigational programming: Orchestrating computations around data. In: PDCS2002: 2002 International Conference on Parallel and Distributed Computing and Systems, Cambridge, MA (2002) 583